ANGULAR 11 + .NET CORE 3.1. REST API - WORKSHOP

INTRO

THE IDEA

This workshop is created to provide a supported introduction to creating web applications in a RESTful manner with .NET Core 3.1. for the backened and Angular 11 for the frontend part. The technologies that we will be using in the new architecture on Fisheries.

During the workshop, we will be creating a simple project to demonstrate the basic tools and concepts used in the mentioned technologies.

DURATION AND SCHEDULE

The envisioned time span of the workshop is two months, and the project will be separated into two major sections (frontend and backend). The sections will consist of weekly tasks and checkpoints. The first four weeks will be designated for introduction to Angular, and the next four to the .NET Core web API. In the end, all of these steps will result in a simple usable web app with simple CRUD and a friendly UI.

In most part of the workshop we will use mock data, and in the end some sample tables will be provided ready for use to finish up the part of connecting the REST API to the DB.

TASKS AND CHECKPOINTS

Every week there will be a short intro of whats coming (on Friday during the tech talk meeting), and then everyone will be given a designated task for the next week ahead. The weekly tasks will be reviewed in more detail on the next Friday meeting. During the week, support will be available for any questions, dilemmas or advice regarding the task.

THE WEEKS - OVERVIEW

WEEK 1 – FRONTEND – SETTING UP THE PROJECT

- Setting up the Angular env
- Installing all necessary things for the setup (VS Code, NODE.js, Angular CLI..)
- Building and running the default test app
- Creating a new component using the CLI commands, displaying the created component in the browser
- Overview of the basic concepts for next time– links to readings + our Angular project setup guide, preparation for the upcoming first task

WEEK 2 – FRONTEND – COMPONENTS, MODULES, TEMPLATES

- Install Angular Material and use it's UI components
- In our new created component template (html) – we are going to implement a few features like a textbox, button, alert pop up, dropdown. and enable some simple actions like „on button click, hide text“ or „on dropdown choice, pop up alert“ etc.
- We are going to try out data interpolation and event binding

WEEK 3 – FRONTEND – DATA BINDING, STRUCTURAL DIRECTIVES, PIPES

- Create a new component to display a list of objects, sort the folder structure a little bit
- Create a class model with some mock data
- Use Material UI table component to create a view for the mocked data
- Try out sorting, paging, pipes, data binding and structural directives on the table

WEEK 4 – FRONTEND – MODELS AND SERVICES, DEPENDENCY INJECTION

- Create a new model class in the Models folder
- Create a new component in the Components folder
- Create a mocked API service on mockapi.io
- Create a Services folder in the project and add a new service class file
- Try fetching the data with an HTTP request, bind it to a table with basic functionalites

WEEK 5 – FRONTEND – INPUT FORM

- Create a new component for the input form for entering a new product
- Create an input form with Material components
- Create a new endpoint in the mock API which will return a list of countries
- Submit the form to the API endpoint

WEEK 6 – BACKEND – SETTING UP THE PROJECT

- Set up the new ASP.NET 5 Web API project in Visual Studio
- Meet the Swagger dashboard
- Investigate the project arhitecture, Startup class etc.

WEEK 7 – BACKEND – ENTITIES, MODELS, MIGRATIONS, DATABASES

- Create entities that will be used to create our database
- Create data transfer objects

- Create the database context
- Add migrations and update database

WEEK 8 – BACKEND - AUTHORIZATION, CONNECTING TO FRONTEND, DATA DISPLAY

WEEK 9 – BACKEND -

WEEK 1

OVERVIEW AND INTRODUCTION

In the first week we are going to set up our project. We are going to install the necessary things like NODE.js and VS Code, and run our first angular app with the `ng-serve` command. We will inspect and alter the content shown in the browser window on our home page. We will also create a simple component in our new app and see it in action in the browser.

We should also cover some initial reading about Angular's core concepts before we continue with other tasks (Check the „LINKS AND READINGS“ section for more detail).

TASK FOR THE WEEK

1. **Installation and project setup**
    a. For the project setup we need to install a few things first: NODE.js, Angular CLI, VS Code – more deatils and how-to are in the links section
2. **Creating and running the app**
    a. After installing all the necesary things, In Visual Studio Code we need to install the Powershell extension by pressing `ctrl + shift + x` inside VS Code (opens the extensions tab) and searching for *Powershell*
    b. After installing the powershell you can choose the folder in which you want to create your new app (like in the terminal), and create a new project by running the `ng new [app_name]` command inside the powershell. You can choose to include Angular Routing or not (choose yes) and choose the stylesheet format (go for the classic CSS), after that your project will be created in the chosen folder and you can open this folder in VS Code to see the files created
    c. By running the app in the browser with the `ng-serve` command trough the powershell, we can inspect the browser view – open the browser and navigate to http://localhost:4200 to see the app

> P.S. you can always stop running the app by pressing `CTRL + C` in the powershell

3. **Creating a new component**
    a. Create a new component through the powershell with the `ng generate component component-name,` and you can see the new component inside the /src folder
    b. Try out the live development through the following steps (when you are running the app, saving changes in the project triger a refresh on the localhost:4200)
    c. Delete everything inside the app.component and add the new components' selector inside the app.component html file (the selector is an HTML tag like `<app-component-name></app-component-name>` used to mark where exactly our components' view will be

loaded in the HTML DOM - you can find the selector specified in the new components' .ts file!)

d. Check out the changes in the browser and play with the new components' html view – change title, create an html <h2> or <p> tags, and print out some text

LINKS AND READINGS

Use the following links to help you with the task on hand:

SETUP

Fisheries Angular project setup guide (chapters 2, 4, 5)

- https://teams.microsoft.com/l/file/1A345ACA-4FB8-4E1C-A06A-E2B06C9003FC?tenantId=eb596cda-cff5-4c5f-ad1a-01db34348dd0&fileType=docx&objectUrl=https%3A%2F%2Fgdigroup.sharepoint.com%2Fsites%2FFisheriesteam981%2FShared%20Documents%2FDevelopers%20corner%2FLearning%20materials%2FAngular%20Project%20Setup%20Guide%2FAngular%20Project%20Setup%20Guide.docx&baseUrl=https%3A%2F%2Fgdigroup.sharepoint.com%2Fsites%2FFisheriesteam981&serviceName=teams&threadId=19:7edf927b833f4be898d444b8cccbe2a4@thread.tacv2&groupId=05e2ef1d-931a-4c0d-a08c-2c31bb7a04e0

Official Angular setup guide

- https://angular.io/guide/setup-local

Pluralsight Angular crash course (project setup part)

- https://app.pluralsight.com/course-player?clipId=f1761ccd-b7ce-48bd-ac25-9110102d7890

PROJECT STUCTURE

Angular file and folder structure explained

- https://www.youtube.com/watch?v=q4fBJtnbqk8&list=PL8p2I9GklV45JZerGMvw5JVxPSxCg8VPv&index=3
- (chapter 6) https://teams.microsoft.com/l/file/1A345ACA-4FB8-4E1C-A06A-E2B06C9003FC?tenantId=eb596cda-cff5-4c5f-ad1a-01db34348dd0&fileType=docx&objectUrl=https%3A%2F%2Fgdigroup.sharepoint.com%2Fsites%2FFisheriesteam981%2FShared%20Documents%2FDevelopers%20corner%2FLearning%20materials%2FAngular%20Project%20Setup%20Guide%2FAngular%20Project%20Setup%20Guide.docx&baseUrl=https%3A%2F%2Fgdigroup.sharepoint.com%2Fsites%2FFisheriesteam981&serviceName=teams&threadId=19:7edf927b833f4be898d444b8cccbe2a4@thread.tacv2&groupId=05e2ef1d-931a-4c0d-a08c-2c31bb7a04e0
- Intro to basic concepts - https://angular.io/guide/architecture

DECORATORS, SELECTORS, ROUTING

- https://www.pluralsight.com/guides/understanding-the-purpose-and-use-of-the-selector-in-angular
- https://angular.io/guide/router

COMPONENTS

More about components:

- https://www.youtube.com/watch?v=iyk98XmsQB4&list=PL8p2I9GklV45JZerGMvw5JVxPSxCg8VPv&index=6
- https://app.pluralsight.com/course-player?clipId=a2752987-674c-466b-ac8c-30f278896c80

- (chapter 7.1., 7.2.) https://teams.microsoft.com/l/file/1A345ACA-4FB8-4E1C-A06A-E2B06C9003FC?tenantId=eb596cda-cff5-4c5f-ad1a-01db34348dd0&fileType=docx&objectUrl=https%3A%2F%2Fgdigroup.sharepoint.com%2Fsites%2FFisheriesteam981%2FShared%20Documents%2FDevelopers%20corner%2FLearning%20materials%2FAngular%20Project%20Setup%20Guide%2FAngular%20Project%20Setup%20Guide.docx&baseUrl=https%3A%2F%2Fgdigroup.sharepoint.com%2Fsites%2FFisheriesteam981&serviceName=teams&threadId=19:7edf927b833f4be898d444b8cccbe2a4@thread.tacv2&groupId=05e2ef1d-931a-4c0d-a08c-2c31bb7a04e0
- Introcution to components and templates - https://angular.io/guide/architecture-components
- Introductin to modules - https://angular.io/guide/architecture-modules

THE FOLLOWING LINKS WILL BE USEFUL READING FOR PREPARING FOR THE NEXT TASK:

More about incoming topics:

- Templates - https://angular.io/guide/template-syntax
- Interpolation - https://angular.io/guide/interpolation
- Binding - https://angular.io/guide/binding-syntax
    - https://angular.io/guide/property-binding
    - https://angular.io/guide/event-binding
    - https://angular.io/guide/template-statements

WEEK 2

OVERVIEW AND INTRODUCTION

In this week we are going to start experimenting a bit more with our newly created application and component. We will install Angular Material in our project, so we can use new UI components like tables, textboxes etc. easily. Material is a library that provides a great out of the box solution for designing nice HTML views.

We will use this design library to add some UI components (these different from the component we created in the last weeks task!) to our new components HTML file.

We will try out data interpolation and propery binding.

TASK FOR THE WEEK

1. Install Angular Material library by running the following command inside the VS Code powershell: `npm install @angular/material @angular/cdk @angular/animations –save`
2. After installation we need to add Material to our application with the command: `ng add @angular/material`. We choose a few options when adding, like a prebuilt theme (choose any you like), global Angular Material typography styles (choose yes) and setting up browser animations (also choose yes). Now we can use Angular Material components in our application
3. When we want to add specific UI components into our HTML, we first need to import their belonging Material modules into our `app.module.ts` file

This file takes care of all our added modules and components across the app. When we want to use a new module or component, we first have to register it there so the app knows about it. Components that we create through the CLI are usually already added there for us, like the first component we've created. The BrowserAnimationsModule – that we chose during the Material installation, has also been added there automatically.

4. For example if we want to add a button with Material styling, we should first import the button module by adding: `import { MatButtonModule } from '@angular/material/button';` to the top of our app.module.ts file where other imports are. Besides that, in the same file we should add the imported module inside the `@NgModule` decorator property called „imports“ (you can't miss in the app.module.ts file, it already has our `BrowserModule` and `AppRoutingModule`!)
5. Then we are ready to use our Material styling – go into our new component's HTML file and add the following tag `<button mat-raised-button>My First Button</button>` - when you run ng-serve in the VS Code powershell, you can see the button displayed in the browser!

You can see more about the Material button tag, and why we used „raised“ on the following link: https://material.angular.io/components/button/overview

P.S. always make sure to (re)build the app with ng-serve AFTER adding new modules, so they can be properly included

In the same manner as adding our button, we can add other Material components like – you can find more about them and how to add and customize them, here: https://material.angular.io/components/categories

6. Besides trying out the Material UI compontents, we are going to try out interpolation and event binding. For interpolating some easy text we should specify a new variable in our new components' .ts file, inside the export class block. E.g. name = 'Pablo' :

```
export class FirstComponentComponent implements OnInit {

  name = 'Pablo';

  constructor() {
  }

  ngOnInit(): void {
  }
}
```

7. With this variable set up, we can use interpolation in our HTML file by adding a piece of code like: `<h1>Hello, {{ name }}</h1>.` The curly brackets will interpolate our defined variable and we can see the results in our browser as „Hello, Pablo“

Read more about interpolaton here: https://angular.io/guide/interpolation and try it out combined with the Material UI components like button, text field, and pop up.

8. Knowing that we can manipulate our HTML from the .ts file, opens the door for many things which we will see later. For example we can define methods in the .ts file that can further expand our data manipulations. For example, we can create a new method in our new components .ts file like this:

```
export class FirstComponentComponent implements OnInit {
  name = 'Pablo';
  constructor() {}
  ngOnInit(): void {}

  OnMyButtonClick() {
    this.name = 'Carlos';
  }
}
```

This method is called a template statement. We can then add the following code into our new components' HTML, where we already have a Material button and our „Hello“ title printed out:

```
<button mat-raised button color= "primary" (click) =
"OnMyButtonClick()"> my button {{ name }} </button>


<h1>Hello, {{ name }}</h1>
```

The (click) event for this button is now bound to our defined method, and we can see the change from „Hello, Pablo“ to „Hello, Carlos“ i our browser when we save the changes.

What we just did here with our method and the buttons' (click) event is called Event binding and you can read more about it here: https://angular.io/guide/user-input, https://angular.io/guide/template-statements and https://angular.io/guide/event-binding.

LINKS AND READINGS

ANGULAR MATERIAL

- https://tudip.com/blog-post/how-to-install-angular-material/
- UI Components - https://material.angular.io/components/categories

INTERPOLATION AND PROPERTY BINDING

- https://angular.io/guide/interpolation
- https://angular.io/guide/property-binding
- https://angular.io/guide/binding-syntax
- https://angular.io/guide/event-binding
- https://angular.io/guide/template-statements

TEMPLATES

- https://angular.io/guide/template-syntax

WEEK 3

OVERVIEW AND INTRODUCTION

This week we are going to be creating a new component that represents a table view for a list of mock data. For this view we will use Material table UI component, with some addtional functionalities like sorting and paging. We'll see how to bind the data to the table, how to use pipes, and how to use a structural directive. We are also going to sort our project folder stucture to start preparing for later tasks.
For the data we are going to create a new class with our test model (entity) called Product, that contains the model definition and a method to return a list of mock data of type Product.

TASK FOR THE WEEK

1. Create a new folder in the /srs/app/, called Components. Move your first component that we created last week into that folder (don't forget to update the import path for the compoment in app.module.ts file)
2. In the newly created folder also add a new component called products-list
3. Create another folder in the /src/app/ directory and call it Models
4. Using the powershell, go inside the Models directory and run the command `ng generate class Product -skip-tests`

This command creates a new Product.ts file in our desired directory. It automatically contains the necessary starting code with the skeleton class block.

5. Inside this file we need to declare some class properties. We will call them: *name, price, category and countryOfOrigin.* Besides the properties, we need a class constructor that will take all these as parameters, and assign them to the object instance.

You can explore typescript basic data types here: https://www.typescriptlang.org/docs/handbook/2/everyday-types.html and general variable declarations here: https://www.typescriptlang.org/docs/handbook/variable-declarations.html

6. Inside the class we need to also create a static method *GetAllProducts(),* that will **create and return** a list of mock data of type *Product* (aim to create about 10 products in the list so we can later implement pagination on our list view)
7. In our new products-list.component.ts file we can now do some modificatins. We need to import the needed material design components and our new Product class, declare the list of Products and call our static method to get the data
8. In the new component, implement the mat-table compoment and bind the data from the list of products as the data source

You can find detailed instructions on how to implement and use the `mat-table` component here https://material.angular.io/components/table/overview. Dont forget to import all new modules in the app.module.ts file as well!

9. After implementing the mat-table component and displaying our list of products, we can add the sort functionality to some or all columns (your choice)

You can find everyting about table sorting in the mat-table overview and examples link above. hint: explore `MatTableDataSource`, Angular lifecycle hooks https://angular.io/guide/lifecycle-hooks and `ViewChild` decorator https://angular.io/api/core/ViewChild

10. Besides sorting our data, we need to implement pagination as a part of our table's functionality . We can do that using the `mat-paginator` component
11. With our functioning table view and functionalities, we will do some more data binding manipulation. Try out using **pipes** and use them to set the *countryOfOrigin* column to be displayed in UPPERCASE, and format the *price* column to be in $.

LINKS AND READINGS

- DATA MODIFIERS (visibility) - https://www.tutorialsteacher.com/typescript/data-modifiers
- TYPESCRIPT CLASSES - https://www.typescriptlang.org/docs/handbook/2/classes.html
- LIFECYCLE HOOKS - https://angular.io/guide/lifecycle-hooks
- MATERIAL TABLE - https://material.angular.io/components/table/overview, https://blog.angular-university.io/angular-material-data-table/
- MATERIAL PAGINATOR - https://material.angular.io/components/paginator/api
- PIPES - https://angular.io/guide/pipes

WEEK 4

OVERVIEW AND INTRODUCTION

This week we are introducing HTTP requests and responses into our project. We've already seen how to simply bind some improvised data to a Material table and create basic functionalities for the table view, and now we are going a step further.

We will use a practical online tool to create mock API with endpoints for our needs that will later be replaced with our real API. We will send an HTTP GET request to the fake API from our Angular app. We will bind the response data to a table and view it in the UI.

TASK FOR THE WEEK

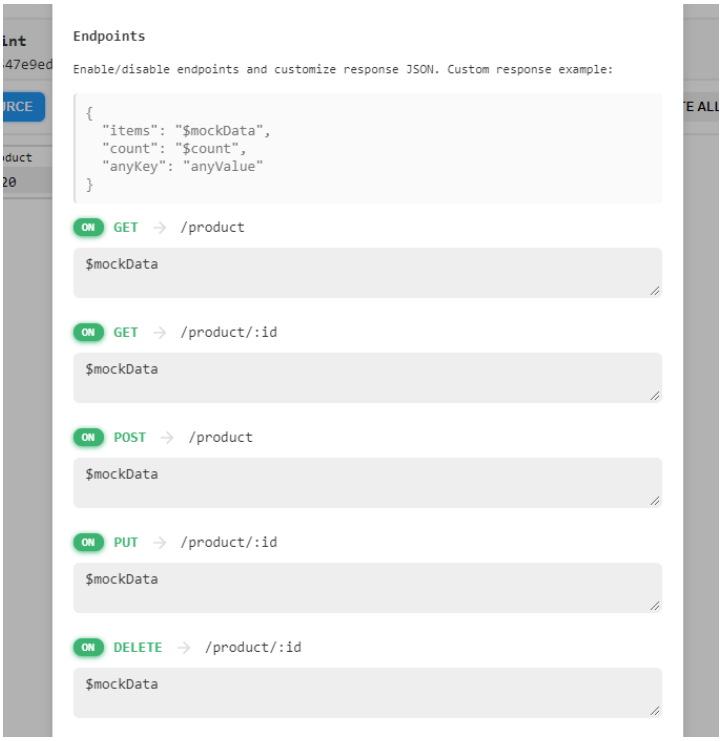
It's best to start by brushing up on HTTP knowledge - https://www.tutorialspoint.com/http/index.htm, https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages, https://developer.mozilla.org/en-US/docs/Web/HTTP/Status - be sure to glance a quick reminder on requests, responses, payload, headers and status codes. If you are already familiar with this enough, feel free to skip this part and carry on.

1. Let's create a new folder called Services, in src/app/. Inside that new folder we are going to create a new service file with the command `ng generate service product --skip-tests`. This creates a product.service.ts file ready for us with some automatic imports.

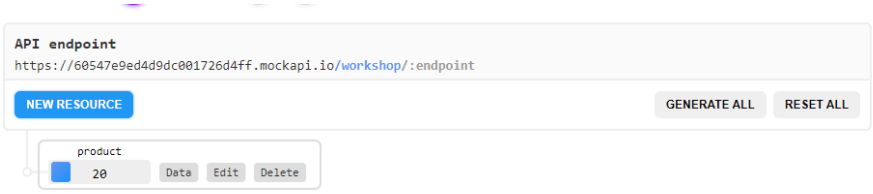
The service file automatically contains the *Injectable* decorator. This allows us to inject our service anywhere in the app. If you are not familliar with it, please discover more on the important concept of Dependency Injection – we will be using it a lot in the upcoming weeks both in Angular and the API - https://angular.io/guide/architecture-services | https://www.freecodecamp.org/news/a-quick-intro-to-dependency-injection-what-it-is-and-when-to-use-it-7578c84fa88f/ | https://angular.io/guide/dependency-injection

2. In our product model file, remove everything but the property declarations, and add an id property
3. Now, let's go to https://www.mockapi.io/ to create our fake API before we continue
    a. Login or create a new account there. You can folllow the Docs instructions to create a simple API with one endpoint. Call the project whatever you like, and add a prefix like *workshop* if you desire
    b. After initial project creation, add a new resource, call it *product* and define the schema – this schema is going to be equal to our product model we already created. Choose Faker.js as a data type and choose what kind of data it should create.
    For now – the path to our resource is going to be defined in the URL as you will see, as *https://<UniqueID>.mockapi.io/workshop/product*

c. This created resource will have some endpoints automatically define. They will correspond the the HTTP method we choose for the request, and the parameters we send in our app. For now you dont have to configure anything there because we are only going to use a GET request to fetch this data – which will correspond to the first endpoint in the image



d. After creating the resource you can choose the amount of data you want generated on a GET request by sliding the blue stripe inside the resource box. If you click the name of your resource here, it wil open a new page with the data objects – you can copy that URL to the service later



For example: a GET request to fetch all products in our app will correspond the URL *https://<UniqueID>.mockapi.io/workshop/product*

Another GET request that fetches just one product by id could be made to the URL *https://<UniqueID>.mockapi.io/workshop/product/:id* where the „:id“ represents the id parameter (int) we are sending from the app.

Now that we've created our fake resource and made it generate some mock data, we can prepare to make a GET request to this fake API endpoint from our app!

4. In our new service file, let's add a method called `getProducts()` which will be of type `Observable<Product[]>` (because the HttpClient's `get()` method we will use, returns an `Observable` type) This method will preform an HTTP GET request to our api endpoint. As you will notice with examples, we need to import the `HttpClient` module, and this time we also need to inject it in the service constructor!

If you haven't read about it yet, here is the dependency injection concept explained in the official Angular docs -> https://angular.io/guide/architecture-services#dependency-injection-di

You can find more about the concept of getting data from a server here - https://angular.io/guide/http#setup-for-server-communication and here - https://angular.io/tutorial/toh-pt6#enable-http-services (skip the in memory data part because we are using our fake API endpoint for this, and also the logging part)

5. Now we can go back to our product-list.component.ts file and do some changes to the existing code. Declare a new array of type `Products[]` if you already dont have it. We will also add a new method called `getAllProducts()` and inside we will call our product service method. To do that, we also must import the product service and inject it in the constructor.
6. Inside the new method, we will call the product service's `getProducts()` method that we created. We will use the in-built `subscribe()` method to fetch our observable result. Inside the subscribe method, you can use a lambda to map the response, something like:

```
.subscribe((response: Datatype) => { this.arrayOfProducts = response; });
```

Find more about subscribing here - https://rxjs-dev.firebaseapp.com/guide/subscription

7. Remove the `ngAfterViewInit()` method, and copy all the code from there into our new method – insdide the subscribe lambda where you map the response to the array. The mat table datasource property should also be set inside this method (still with the MatTableDataSource())
8. If you dont have it, add an `ngOnInit()` method, and inside it, call the `getAllProducts()` method. We should now be all set and done, and our table should be bound to the data getched from the fake API.

LINKS AND READINGS

- SERVICES - https://angular.io/tutorial/toh-pt4
- SERVICE PROVIDERS - https://angular.io/guide/providers
- DEPENDENCY INJECTION - https://angular.io/guide/architecture-services#dependency-injection-di https://angular.io/guide/dependency-injection
- LAMBDA FUNCTIONS - https://www.tutorialsteacher.com/typescript/arrow-function
- RXJS SUBSCRIBE METHOD - https://rxjs-dev.firebaseapp.com/guide/subscription
- MOCK API DOCS - https://www.mockapi.io/docs
- HTTP - https://angular.io/guide/observables-in-angular#http, https://angular.io/tutorial/toh-pt6#httpclient-methods-return-one-value

WEEK 5

OVERVIEW AND INTRODUCTION

This week we are going to create an input form to add a new product into our imaginary database via the mock API we already created.

Firstly we will create a new component for our input form, create two input fields and two options fields using Material components. The input will correspond to our existing Product model. We will also add a new endpoint to our mock API that will return a list of country names only, so we can use that in our options field. The categories list is also needed, but we will do that in our .ts file to show how it can be done there.

After submitting the form data, we will be sending a POST request to our mock API, to add our new product to the list of existing products, where we will be able to see our product listed.

TASK FOR THE WEEK

1. Create a new component called insert-product
2. Before we continue, lets add a new endpoint to our mockapi.io API we created last time. It should still be available when you log in. Click *NEW RESOURCE* and create an endpoint with just the id and country parameters. Choose the option to generate names of countries. Set the number of countries you want returned with the blue slider. Take note of the url that leads to your endpoint
3. In our ProductService.ts file, let's add a `getCountries()` method, and send a GET request to our new endpoint, just like we did with products last time. The only difference is that, in this case, we need to define an anonymus return type list. If you don't want to do that, you can also create a `Country` class model in the models folder and use that instead. However it's interesting to see that we can use anonymus data types as well.

> Find out more about anonymus data types here:
> https://www.typescriptlang.org/docs/handbook/2/objects.html , they are somewhat simmilar to C#.

4. Now, let's get back to our form. Use Material components that can be found in the official documentation to create an input form in HTML (mat-select, mat-form-field, mat-label, input, form, button etc..).

> It is strongly recommended to scan our Slatkovodno project code for keywords and form examples while doing this task. There are a lot of examples of input forms that use Material design.

The form should contain two input fields – one for text (name of product), and the other one for number (price of the product). We also need two options fields (dropdowns), one for categories and another for the country of origin

5. Now lets configure the data resources for our options lists. In the product-insert.component.ts file, create a list of strings and add some category names to it. We will use this for our list of categories dropwdown. If you want, to keep the data sources completely separate from the component, you can also define this as a method in the product service file, next to our GET requests.

> Also when you are using the service methods in the component file, dont forget to inject the ProductService into the constructor and create lists of objects of the same type as what the methods return. Fetch the data in the same way as we did last time with products and save it in the lists

6. Prepare and initialize the necessary form controls in the .ts file
7. Now that we have our input data, we need a way to save it to our products list. In the ProductService.ts file, create a method called insertProduct(). In that method we will do a POST request and send our data to the API – again you can check our Slatkovodno project for references to POST methods if you get lost
8. Add an onSubmit() method in the component file to handle and save our form input values after submitting data
9. Call the insertProduct() method inside the onSubmit() even and pass it the form values object to transfer the form data to the service endpoint
10. After that, go back to your list of products, and try to locate your newly added product!

LINKS AND READINGS

- FORMS https://angular.io/guide/forms, https://angular.io/start/start-forms
- MATERIAL FORMS https://material.angular.io/components/form-field/overview , https://material.angular.io/components/input/overview , https://material.angular.io/components/select/overview
- FORM CONTROL AND FORM GROUP https://blog.logrocket.com/reactive-form-controls-form-groups-angular/
- Slakovodno frontend project is located in our HR_MPUR_3.0. repository https://gdigroup.visualstudio.com/GDi%20IISWE/_git/HR_MPUR_3.0?path=%2FSlatkovodno%2FFrontend

WEEK 6

OVERVIEW AND INTRODUCTION

In this week, we are switching to the backend. We are going to create a web API project in visual studio, using the .NET 5 framework. We will check out the structure of the API architecture that we've created and set up our local database. We will see how the project is configured, how the database is connected and how we can set up different environments. We'll check out Swagger – an integrated tool for sending requests to our API and inspecting responses from the endpoints.

TASK FOR THE WEEK

1. We'll start by opening up Visual Studio 2019 and creating a new project. Select the ASP.NET Core Web API template, choose a name, locate it in the same root folder as your frontend Angular application. When prompted, choose .NET 5.0. as the target framework. You can leave other options on default for now

> .NET 5.0 is the latest version of the .NET Core framework. If is not offered in the dropdown as the target framework be sure to download and install the appropriate SDK from the official website.

2. Once we have created the project, you will see a the project structure in the Solution Explorer. There will be an example WeatherForecast Class and Controller already created. Besides that the most imporatnt files we should take notice of are the following:
    - Startup class – here we will define all of our application services, configure the request procesing pipeline and basically do anything and everything that we need to have when we start our application.

    > Check out more detailed information about the construction of the startup class here: App startup in ASP.NET Core | Microsoft Docs

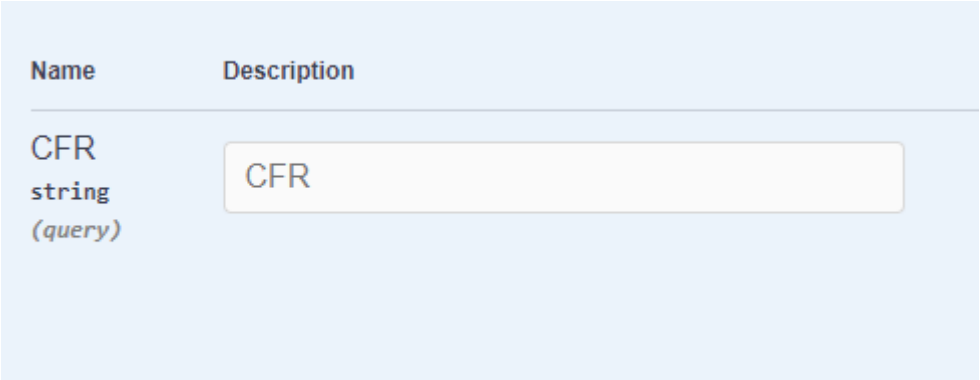
    - Program class – this is the place where the host is built and the Startup class is specified and called
    - Appsettings.json and appsettinigs.Development.json files – here we configure our connection strings, allowed hosts, logging etc
    - Properties/launchsettings.json file – here we configure our environments, launch profiles etc

    Besides these files, the Controller folder is pretty important to us, because here all of our application controllers will be located by convention. They contain our endpoints – the entrance to our application from the outside world. There are other things to be noted, and many more specifics of the REST API architecture, but we are not going to cover them all here in this document.

3. Now that we've seen some basics about our new project, let's try to run it. Choose the project name from the run dropdown optrions, instead of IIS Express. When the browser opens, it will take you directly to the Swagger page. Swagger is a cool tool that comes already set up with the latest version of the .NET framework. All methods you see in Swagger correspond to the methods defined in your controller classes. They are labeled by the name you provided to the methods in the controller and they are marked with the HTTP method type that you defined in the method itself.

This is great because it not only allows you to easily test the capabilities of the API without the need to use an external tool like postman but it also documents the API so that each time you, or someone else looks at it they can get a feeling of all the actions it provides.

All of the methods that you can see in the Swagger UI dashboard host a lot of parameters which are parameters that correspond to the parameters that the controller method has.



After you finish typing in parameters (if the method has any) you can press the „Try it out“ button to get the output from the API.

4. Now that you got to know where the basic files are, what is their basic purpose we will go through the process of structuring your API in a way that conforms to a very popular architectural standard the Controller-Service-Repository pattern. I will go through each of those layers and explain what each of them do and what you should write into them.
    - **Controller Layer**
      You have already seen the controller layer because it gets created by default once you create your .NET 5 web API project. This layer is the only necessary part of each application because the application wouldn't work without it. The files placed in this layer, the controllers, are there to expose the capabilities of the API to the client that consumes it through various methods. Because of that, these files should only contain one to two lines of code. The methods should be clean and all of the logic should be hidden away from them and placed into its own layer, the service layer.
    - **Service Layer**
      The service layer holds the main logic of our API. This is where all the magic happens. We receive some parameters in the controller layer, and the controller passes those parameters to the service layer where we do something with them. We group the data to complex objects, we log information to the log, we throw errors and all sorts of stuff. The one thing we don't do in the service layer is insert,update,delete,and retrieve data from the database. We have the repository layer for that.
    - **Repository Layer**
      The repository layer is where our context class is injected. We will discuss more about the context class in the next week. But the most important thing to note here is that you write all your database queries in the repository class. This is where you retrieve, insert update and delete data from the database.

READ MORE: The service is injected in the controller (one controller should only have one service), and the repository is injected in the service with the dependency injection software design pattern. Read more about it here Dependency injection in ASP.NET Core | Microsoft Docs.

This pattern is not set in stone. You don't actually need to have 3 layers for the application to work. If we agree to it, we can completely remove the repository pattern. The repository pattern is entirely optional but the controller and service layer are necessary in a real world application. The logic should never be contained in the controller.

After reading these basic informations we can start creating our API. First of all we will create a new controller.

The tasks:

1) Delete the WeatherCastController class and create a new controller you will name ProductsController.
2) Create a new HttpGet method in the controller you will call GetProduct.
3) Create a new service class you will call ProductsService in the folder Services (you will need to register the service in the Startup class)
4) Inject the newly created service with dependency injection in the controller.
5) Create a new method GetProduct in the service class which will return a string (give it some random value)
6) Call the service GetProduct method in the controller GetProduct method.
7) Run the application and go to the swagger url.
8) Test the method

Congratulations, you have created your first API endpoint.

LINKS AND READINGS

- STARTUP CLASS App startup in ASP.NET Core | Microsoft Docs
- SWAGGER UI - REST API Documentation Tool | Swagger UI
- CONTROLLER – SERVICE – REPOSITOR PATERN The Repository-Service Pattern with DI and ASP.NET Core (exceptionnotfound.net),
- DEPENDENCY INJECTION Dependency injection in ASP.NET Core | Microsoft Docs

WEEK 7

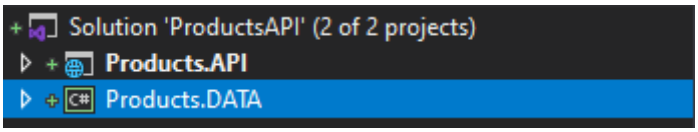
OVERVIEW AND INTRODUCTION

This week, we are creating the data layer of our API. We will use the already familiar Entity Framework to create a database using the standard Code-First approach. While Database-First and Model-First are approaches which were heavily used in the past and were endorsed by Entity Framework, since the 4.1. version, the guys behind entity framework are moving the framework completely in the Code-First direction. Once we create our local database we will go over the appsettings file and how we can connect to a development and production databases and will create migrations to update the state of our database through code! That will cover most of the basic things you need to create a database and use it an effective way.

TASK FOR THE WEEK

First of all, we are going to create a new project in our solution that will contain all of the models, entities and the context class and everything else that should logically go into the data layer of an application.

1) Create a new class library, add it to the solution and add a project reference from your main API project to that newly created class library.



2) Once we create the class library we have to install entityframework. Install the EntityFrameworkCore, EntityFrameworkCore.Relational and EntityFrameworkCore.SqlServer packages to the DATA project and EntityFrameworkCore.Tools and EntityFrameworkCore.Design to all projects in the solution.
3) Next up we will create the entities which will get translated into SQL tables. Create a new Folder called Entities and create two new public classes inside. Name one of the classes Product and the other ProductType.
4) Add properties to those classes. One property should be of type int and it should ideally be called Id or ProductId. After that add a Name property, a Price property and a Type property which will be of type ProductType. For the TypeProperty add a foreign key, TypeId. This is not necessary but if we want to seed data easily, it makes it easier. Lastly, add a Country property of type string. In the ProductType class add the Id property and a Name property.
5) In the same folder create a new class called ProductDbContext. It should inherit from the EntityFrameworkCore.DbContext class. Also create a constructor and make it inherit the base properties by using the base() syntax. Lastly, create two properties of type DbSet<Product> called Products and a second of type DbSet<ProductType> called ProductTypes.
6) In this step we are returning to our API project and the appsettings.json file. We have to tell our application what is the connection string to our database and we also have to register the DbContext class in the Startup file. Remember, the Startup file is the heart of our application, you've been reading about it in the previous week and you have a link there to refresh your memory. In the appsettings file add a line similar to this:

```
"ConnectionStrings": {
    "DbContext":  "Data source=localhost\\SQLEXPRESS;Initial
Catalog=ProductDb;Trusted_Connection=True"
}
```

And to register the context class, add the following lines in the ConfigureServices method of the Startup class.

```
services.AddDbContext<ProductDbContext>(options =>  options.UseSqlServer(Configurat
ion.GetConnectionString("dbContext")));
```

7) After we've done all this we can go to the package manager console.

IMPORTANT!: For this step you will need to set up localdb on your machine. If you didn't already, you can follow this tutorial to do so: How to install Microsoft SQL Server Express LocalDB (sqlshack.com)

In the package manager console, type Add-Migration InitialMigration. This will create a Migrations folder where you can see all your migrations. If you open your InitialMigration.cs file you will see it has two methods. The Up method, which indicates what will happen if you Update the database with this migration and the Down method which will execute if you restore the migration to a previous state. You can read more about EFCore Migrations here: Code-based Migration in Entity Framework (entityframeworktutorial.net).

HINT: You can also use the .NET Core CLI but I won't cover that in this tutorial. If you would like to use the .NET Core CLI instead of the Package Manager Console in Visual Studio, you can read up on it here: EF Core tools reference (.NET CLI) - EF Core | Microsoft Docs.

Now it is time to type Update-Database in the package manager console. This will execute all migrations you created, one by one, and update the database. If you open your SSMS and connect to your local database, you will see your database with two tables: Products and ProductTypes. Notice that the Id is autoincrement and

that the Product table has a foreign key to the product types table even though you didn't specify that yourself. It was all inferred by EntityFramework.

8) We will now seed some data to our database. To seed data add these lines of code to your dbContext class:

```
0 references | 0 changes | 0 authors, 0 changes
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  modelBuilder.Entity<ProductType>(entity =>
  {
    entity.HasData(
      new ProductType { Id = 1, Name = "Kitchen Utensils"},
      new ProductType { Id = 2, Name = "Transportation"},
      new ProductType { Id = 3, Name = "Jewlery"},
      new ProductType { Id = 4, Name = "Consumables"},
      new ProductType { Id = 5, Name = "Furniture"}
      );
  });

  modelBuilder.Entity<Product>(entity =>
  {
    entity.HasData(
      new Product { Id = 1, Name = "Spoon", Price = 32.99M, ProductTypeId = 1, Country = "COLOMBIA" },
      new Product { Id = 2, Name = "Bicycle", Price = 569.99M, ProductTypeId = 2, Country = "CROATIA" },
      new Product { Id = 3, Name = "Fancy necklace", Price = 1600.00M, ProductTypeId = 3, Country = "ITALIA" },
      new Product { Id = 4, Name = "Bottle of water", Price = 4.5M, ProductTypeId = 4, Country = "FRANCE" },
      new Product { Id = 5, Name = "Chair", Price = 260.00M, ProductTypeId = 5, Country = "ITALIA" }
    );
  });
}
```

After you are done, add a new migration, call it SeedData or something similar and UpdateDatabase once again. Congratulations, you now have data in your database.

9) Lastly, just create a new folder called Models and create a new class called ProductDTO there. Add all properties to that object that the entity has without the id, for the purpose of this tutorial we will not send the id to the client. That is why DTOs are used. We don't want to send all data the entity has back to the client. That's why we create DTO's and BM's to send specific data.

That's it! We will use our newly created DTO in the next week to send some data back to the client.

LINKS AND READINGS

- EF CORE CODE-FIRST What is Code-First? (entityframeworktutorial.net)
- EF CORE TOOLS CLI - EF Core tools reference (.NET CLI) - EF Core | Microsoft Docs
- EF CORE TOOLS PACKAGE MANAGER - EF Core tools reference (Package Manager Console) - EF Core | Microsoft Docs
- EF CORE MIGRATIONS - Migration in Entity Framework Core (entityframeworktutorial.net)

WEEK 8

OVERVIEW AND INTRODUCTION

This week we will create models (called data transfer objects or DTOs) to transfer data back to the client. We will query the data from our database in the repository layer and map that data to DTO's in the service layer. We will then return the DTO through a new API endpoint. This will be the basic workflow of almost any API that implements a basic CRUD functionality. In the next week we will then connect our API to the previously created Angular application which will conclude this tutorial.

TASK FOR THE WEEK

1) First create a new folder called Repositories and inside create a ProductRepository and an interface for that repository IProductRepository. Register the repository just like you registered the service in the Startup file.
2) Create a constructor in the repository and inject the dbcontext class with dependency injection.
3) Create a new method in the repository called GetProducts which returns the DbSet from the context class which we name Products.
4) Inject the repository with dependency injection in the previously created service class.
5) Create a new method in the service class and call it GetProducts as well but this time this method returns IEnumerable<ProductDTO>.
6) Now we will use automapper in order to avoid explicitly mapping the product entity to the product dto. First install automapper and automapper.dependencyinjection using the nuget package manager. After that you can create a new folder named Configuration and add a new class called MappingConfiguration there. Every time we add a new DTO or a new Entity we create the mapping logic here. The is a lot of custom mapping you can write here and you can read more about it on the official automapper documentation: AutoMapper. I won't go into detail how to map two objects but there is a good explanation you should find in the Automapper documentation. It is just a line or two of code. The mapper automatically detects if two variables have the same name and maps them accordingly. Don't forget to register AutoMapper in the startup class!
7) In our method it is time to map the incoming list of products to a new list of product DTOs. You can do so by simply writing:

```
0 references | 0 changes | 0 authors, 0 changes
public IEnumerable<ProductDTO> GetProducts()
{
    return mapper.Map<ProductDTO[]>(productRepository.GetProducts());
}
```

8) Now lets create a new controller method in our controller. It will also be of type GET because we are retriving data. Because we have two HTTP get methods now, we must name our method explicitly. You can do that by providing a string in the HttpGet attribute like this:

```
[HttpGet("GetProducts")]
0 references | 0 changes | 0 authors, 0 changes
public IActionResult GetProducts()
{
    return Ok(productsService.GetProducts());
}
```

9) It is time to test our method now! Run the application and try out the new method 😊

LINKS AND READINGS

- AUTOMAPPER - AutoMapper
- RESTful APIs - What is REST API (RESTful API)? (techtarget.com)

WEEK 9

OVERVIEW AND INTRODUCTION

This week we are going back to Angular! We are finally going to connect our API and the Client. We will create a service class in the client. Ideally you would have one service class per one controller when you are using the

.NET Core Angular stack. In that service we will create a method that goes to the API endpoint and asks for the data. We will then display that data in the grid and delete the hardcoded data that we created previously.

TASK FOR THE WEEK

1) First of all, go to the angular project environments folder and add a new property to the environment constant with the url of our API.

```
appUrl: 'https://localhost:44303/'
```

2) After that create a new folder called services under src/app and with the angular cli create a new service by typing „ng generate service products“. Inject the HttpClient in the constructor, declare two variable called appUrl and apiUrl of type string and initialize them to correct values in the constructor. Besides that create a new variable and assign HttpHeaders to it where you will define the content type and charset:

```
httpOptions = {
  headers: new HttpHeaders({
    'Content-Type': 'application/json; charset=utf-8'
  })
};
```

3) Create a method called getProducts() which returns an Observable of Product array which you created in the early sections of this tutorial in the models folder.

> **READ MORE:** To read more about Angular Observables and why they are necessary in this situation, you can read more about it here because it is out of scope for this tutorial: Angular - Observables in Angular.

Remove the getProducts method from the models object because a model should never contain any logic in a real world application. That was just for tutorial purposes.

4) In the getProducts method write this line of code:

```
return this.http.get<Product[]>(this.appUrl + this.apiUrl);
```

5) Now we have created our service and the method to retrieve products from our API. We now have to insert those products in our table. In the ProductListComponent remove the method call where we initialize the dataSource variable and leave it blank (just new MatTableDataSource()). In Angular, everything that is required to load during initialization should be done in the ngOnInit() method.
6) In the constructor inject our service with dependency injection, similar to how it is done in .NET but you don't have to assign the value to a field. You can simply use it by using the „this“ keyword.
7) In the ngOnInit method call our method and subscribe to its value (because we are passing an observable and have to subscribe to it to access its value) and create a new instance of MatTableDataSource but this time provide the result as the parameter like this:

```
ngOnInit(): void {
this.productsService.getProducts().subscribe(products => {
   this.dataSource = new MatTableDataSource(products);
 });
```

> **IMPORTANT!** Don't forget to add your newly created service in the providers section of app.module and the HttpClientModule in the imports section of the same module.

8) Now we have set up the angular part, we have to return to our API to do one last thing before we hook everything up and that is enable CORS. I won't go in a lot of detail regarding CORS but it is something you need in order to allow a JavaScript application to make an API call to our API. In the Startup file add this code to the ConfigureServices method:

```

services.AddCors(options =>
```

```
{
  options.AddPolicy("CorsPolicy",
    builder => builder.AllowAnyOrigin()
    .AllowAnyMethod()
    .AllowAnyHeader());
});
```

and also this code to your Configure method:

```
app.UseCors("CorsPolicy");
```

That is it! We have successfully connected and created our .NETCore/Angular application where we retrieve some products and display them in the client application.

Name	Last Name	Age	Country
Spoon	$32.99		COLOMBIA
Bicycle	$569.99		CROATIA
Fancy necklace	$1,600.00		ITALIA
Bottle of water	$4.50		FRANCE
Chair	$260.00		ITALIA

IMPORTANT BONUS TASK! As you might have noticed, the category didn't map correctly and the value that is being sent is null. That is because we need to retrieve it by its foreign key before mapping it to a DTO. This is a bonus task for you to create a new method that finds the product category by its ID in the repository and then map that value to the object before sending it in the service class.

LINKS AND READINGS

- CORS - Cross-Origin Resource Sharing (CORS) - HTTP | MDN (mozilla.org)
- ANGULAR SERVICES - Angular - Add services